CS4605/Lab 4

George W. Dinolt

August 17, 2004

1 introduction

There are two goals of this lab. The first is to show how to define a "state" and the "transform functions" of a "system" to move from one state to the next. The other is to illustrate the layering/abstraction approaches that we will use to describe many of the policies that we will discuss.

To accomplish these goals, we will extend the `seq_theory` that we used in Lab 3. We will do this by "instantiating" the parameters of the theory. Our result will be a proof that the particular seq that we define is **secure** with respect to the definition of security we provided.

This new theory, `triv_state`, is still very abstract. As we shall see, there are still a number of undefined types and operations associated with the theory.

2 Description of the Theory

You can find the theory either here or on `proof` in
`/disk1/cisr/pvs-examples/lab4/triv_state.pvs`. [1]

In the sections below, we provide some description of the various parts of the theory.

2.1 The Parameters of the Theory

In Figure 1 we show the paramaters the we need to describe the system. Any *implementation* of the system will have to provide these parameters.

[1]This is a *hot-link* that points to
http://www.nps.navy.mil/cs/Dinolt/Courses/AY2004/Summer/CS4605/Labs/lab4/triv_state.pvs

triv_state[T: TYPE+, inNets: TYPE+, outNets: TYPE+, SL: TYPE+, read:
[T $\rightarrow$ inNets], write: [T $\rightarrow$ outNets], slIn:
[inNets $\rightarrow$ SL], slOut: [outNets $\rightarrow$ SL]]: THEORY

Figure 1: The Parameters for the triv_system

T is the type of element that is read from $inNets$ and written to $outNets$

$inNets$ is some way of describing the input networks. We do not ascribe any sequencing properties to these at this level

$outNets$ is similar to $inNets$

$read$ is a function that associates a particular $inNet$ with an element.[2]

$write$ is a function that associates a particular $outNet$ with an element.[3]

$slin$, $slout$ associate a security label with each input and output network.

An interesting observation about this theory is that there is no definition of the "meaning" of either $read$ or $write$. Of course we have no notion of how the networks are implemented, only that each element was $read$ from a uniqe network and $written$ to a unique network.

2.2 The Definitions and Internal Lemmas

In Figure 2 we show how the $State : Exp$ is defined, what $st?$ means on the $State : Exp$ and how $transform$ works.

The $State : Exp$ is modeled as a "tuple" or "record" that consists of

- An element of T to be processed, the iu,[4]
- The input nework, $inputNet$ that is part of this state and

[2]As a consequence of this definition, each element of T is asociated with exactly one input network.

[3]Here again, each element of T will be written out on exactly one output network.

[4]The name iu is used for *Information unit*, which is a name inherited from other projects I have worked on.

```
BEGIN

 State: TYPE =
 [# iu: T,
    inputNet: inNets,
    outputNet: outNets #]

 i: VAR inNets

 x: VAR T

 st: VAR State

 transform(x, st): State =
      IF slIn(read(x)) = slOut(write(x))
         THEN (# iu := x,
                  inputNet := read(x),
                  outputNet := write(x) #)
      ELSE st
      ENDIF

 st?(st: State): bool =
      slIn(st`inputNet) = slOut(st`outputNet) ∧
        st`inputNet = read(st`iu) ∧
         st`outputNet = write(st`iu)

 transformSecure: LEMMA
    st?(st) ⇒ st?(transform(x, st))
```

Figure 2: The Definitions for the triv_state Specification

- The output network, $outputNet$ that is part of this state.

The var's that are defined, i, x, and st, are there so that I don't have to use $forall$[5] in the definitions and lemmas below. Whenever a variable appears in such places, it means that the entity is defined for every value of the variable.

The definition of the $transform$ function is relatively straightforward. For any element $x \in T$ and any $st \in State$, we "transform" the st to a new $State : Exp$ by following the rules. If the security label of the $inputNet$ is the same as the security label of the $ouputNet$ then construct a state element consisting of the element x, the $inputNet$ and the $outputNet$. Otherwise use the old state. Note, that the $State : Exp$ does not normally depend on past states.

The definition of the function $st?$ (secure state ?) is similarly straightforward. We ensure that the element was $read$ from the $inputNet$ and $written$ to the $outputNet$ and that the labels of the neworks agree.

You have to prove the $transformSecure$ lemma, that the transform of a secure state is secure.

2.3 Defining a Sequence and the Layering

Finally we get to the gist of the matter, illustrating the layering techniques. This is shown in Figure 3. To use `seq_theory`, we have to have a sequence. But to get that, we need a sequence of elements of T to process. $InputSeq$ is such a sequence. Note that it specifies some unbounded sequence of elements of T. We don't know what the elements are, but we don't really care.

The sequence of $State : Exp$s is now defined in terms of the sequence of elements of T. The goal of the function $seqState$ is to define such a sequence. We do it recursively using the $transform$ function defined above. The **MEASURE** is required to show pvs which variable is involved in the recursion.

Finally, we have to establish that $seqState(0)$ is secure. We do this the easy way by just *assuming* it, using the **AXIOM** command of **PVS**

We now get to the laying concept. The *IMPORTING* command of **PVS** allows us to import another specification. You can look up the command in the **PVS Language Reference Manual** that you can downlowd from their web site. In the *IMPORTING* line we reference the `seq_theory` and each of its parameters. You should verify that the parameters are correct and of the right type.

Since we are importing `seq_theory` we must ensure that each of its assumptions is met by the parameters that we are substituting. When you generate

[5] Remember the mathematical symbol for this is "$\forall$."

```
  InputSeq: sequence[T]

  seqState(n: nat): RECURSIVE State =
    IF n = 0
      THEN (# iu := InputSeq(0),
              inputNet := read(InputSeq(0)),
              outputNet := write(InputSeq(0)) #)
    ELSE transform(InputSeq(n), seqState(n - 1))
    ENDIF
     MEASURE n

  InputSeq_0_secure: AXIOM st?(seqState(0))

  IMPORTING seq_theory[State, T, seqState, st?, transform]

  InputSeq_secure: THEOREM
    every(st?)(seqState)

END triv_state
```

Figure 3: The Lemmas and the Layering in triv_state

```
Proof summary for theory triv_state
   transformSecure......................untried             [Untried]( n/a
   seqState_TCC1........................proved - complete   [shostak](0.56
   seqState_TCC2........................proved - complete   [shostak](0.03
   IMP_seq_theory_TCC1..................unfinished          [shostak](0.12
   IMP_seq_theory_TCC2..................proved - incomplete [shostak](0.18
   IMP_seq_theory_TCC3..................proved - incomplete [shostak](0.58
   InputSeq_secure......................untried             [Untried]( n/a
   Theory totals: 7 formulas, 5 attempted, 4 succeeded (1.47 s)
```

Figure 4: The output of the `ESC-x prove-tccs-theory` command

the $tccs$ for the `triv_state` theory, you will find out that each of the assumptions from `seq_theory` becomes a tcc for `triv_state`. For the importation to be correct, we need to be able to prove each of these assumptions. In this case, the proofs are trivial (almost).

Once we have imported `seq_theory`, the final theorem is literally a triviality.

3 The Actual Details of the Lab

3.1 The steps of the labs

The actual details of the lab are relatively straightforward. You should obtain the pvs file for `triv_state` either from the web site or from `proof` as described above. You should place the file in the same directory that you have the `seq_theory` specification.

You need to ensure that you did the proofs of `seq_theory` in this directory. If you didn't, you should redo them now.

You should generate the $tccs$ for `triv_state` and the command `ESC-x prove-tccs-theory`[6] to try and prove the **tccs** for the theory. The output should like similar to that shown in Figure 4. You will find that all but one of the will be proved. Only one will be "unfinished." The steps below can be used to prove the last one. Make sure that your cursor is in the pvs file and issue the `ESC-x tccs` command. The output you see should be similar to that shown in

[6]Remember that the notation `ESX-x` means type the ESC key and then the "x" key. Ignore the "-"

Figure 5. Move your cursor into the $tccs$ buffer and to the area where the unfinished tcc is located. You should note that it is an assumption about $seqState(0)$. Issue the command `ESC-x prove` and attempt to prove this using (note the verb here) anything you might know about $seqState(0)$ from the specification.

Once this proof is completed, issue the `ESC-x tccs` command again, from within the `triv_state.pvs` buffer, to see that all the tccs are now marked "completed," even the ones that were marked "incomplete." You should study the tccs. Note that 3 of them correspond to the *assumptions* that must be satisfied by, `seq_theory`.

You shoud now prove the $transformSecure$ lemma. You can prove this using only the commands

```
skolem!, expand, flatten, and split
```

If you find yourself with equations in the consequent (below the line), you may want to try the command `lift-if`. There is a somewhat longer proof that uses this command. The lemma itself, is actually pretty trivial.

Finally you need to prove the theorem $InputSeq\_Secure$. The only thing you need to do is to use the theorem from `seq_theory`.

3.2 What you will turn in

After you have completed the work described above, you should run the command `ESX-x show-proofs-importchain` while your cursor is in the `triv_state.pvs` buffer. You should insert your name and the lab number in the top of the buffer, save it, print it and hand it in. The output will show a form of the proofs (in lisp notation) and the state of the theory.

Good Luck

```
% Subtype TCC generated (at line 33, column 41) for  n - 1
    % expected type  nat
  % proved - complete
seqState_TCC1: OBLIGATION FORALL (n: nat): NOT n = 0 IMPLIES n - 1 >= 0;

% Termination TCC generated (at line 33, column 32) for  seqState(n - 1)
  % proved - complete
seqState_TCC2: OBLIGATION FORALL (n: nat): NOT n = 0 IMPLIES n - 1 < n;

% Assuming TCC generated (at line 39, column 12) for
    % seq_theory[State, T, seqState, st?, transform]
    % generated from assumption seq_theory.seq_0_secure
  % unfinished
IMP_seq_theory_TCC1: OBLIGATION st?(nth[State](seqState, 0));

% Assuming TCC generated (at line 39, column 12) for
    % seq_theory[State, T, seqState, st?, transform]
    % generated from assumption seq_theory.transition_state_secure
  % proved - incomplete
IMP_seq_theory_TCC2: OBLIGATION
  FORALL (st: State), (x: T): st?(st) => st?(transform(x, st));

% Assuming TCC generated (at line 39, column 12) for
    % seq_theory[State, T, seqState, st?, transform]
    % generated from assumption seq_theory.seq_transform
  % proved - incomplete
IMP_seq_theory_TCC3: OBLIGATION
  FORALL (n: nat):
    EXISTS (x: T):
      nth[State](seqState, n + 1) = transform(x, nth[State](seqState, n));
```

Figure 5: The output of the `ESC-x tccs` command after trying to do the proofs.